Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

CONTI

FEATURES

OBJECT PERSISTENCE: BEYOND SERIALIZATION 19
by Timo Salo, Justin Hill, Scott Rich, Chuck Bridgham, and Daniel Berg
Our authors describe techniques and frameworks necessary to successfully implement scalable object persistence for complex database systems. Much of the technology they examine has been incorporated in development tools ranging from VisualAge for Java, to EJB tools for WebSphere.

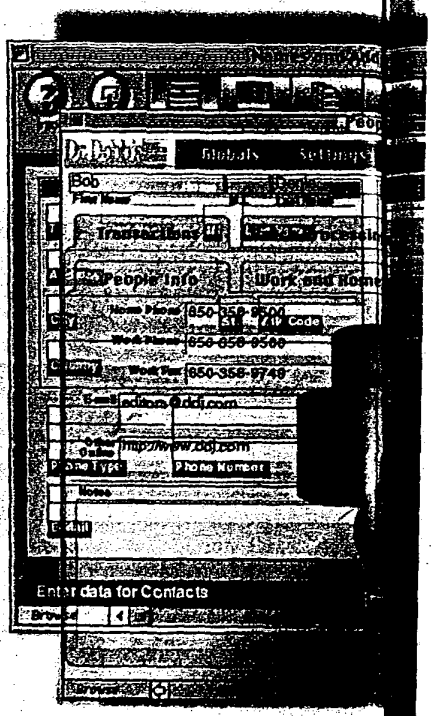
JAVA PROXIES FOR DATABASE OBJECTS 34
by Paul Lipton
Java proxy technology lets you define database object schema using the database ODL. To illustrate how such a technology might be implemented, Paul provides examples based on the Jasmine object-oriented database.

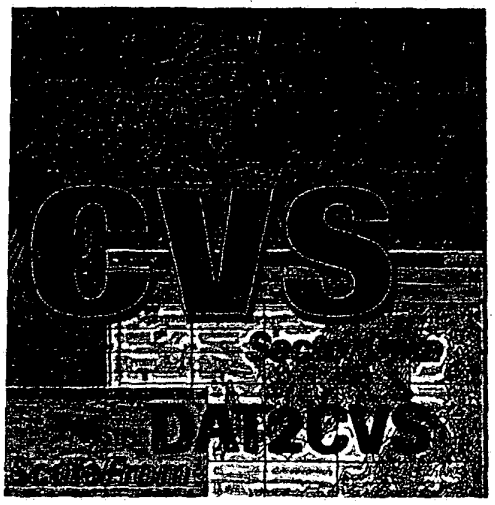
VBSCRIPT AND SQL CALENDARS 40
by John Donovan Lambert
John presents the VBScripts he uses for inputting SQL results into a web calendar, and discusses how you can port these scripts to Java, Perl, Cold Fusion, or whatever language you prefer.

THE CVS DATA FORMAT 50
by Cesar A. Gonzalez Perez
The CVS data format stores cartographic data for a specific geographic area into a single file. Cesar examines the format, then presents a tool for converting CVS files into DXF format.

AGENT ITINERARIES 60
by Russell P. Lentini, Goutham P. Rao, and Jon N. Thies
Instead of examining itineraries in the traditional way as a list of tasks to be performed by agents, our authors treat itineraries as a metaprogram—a way of programming an agent and inadvertently its goal. To illustrate, they'll present an itinerary that performs a database query.

JAVA AND DIGITAL IMAGES 72
by David H. Martin and Johnny Martin
Capturing, storing, and retrieving images is an often-overlooked feature that many applications could benefit from. David and Johnny describe "Grabber for Java," an API that encapsulates the functionality necessary for video capture.





EMBEDDED SYSTEMS

THE SPARK REAL-TIME KERNEL 80
by Anatoly Kotlarsky
SPARK, short for "Small Portable Adjustable Real-time Kernel," is a royalty-free, fast, tiny, portable real-time kernel. Anatoly describes how he used it to build a video bar-code scanner.

INTERNET PROGRAMMING

AUTOMATED TESTING FOR WEB APPLICATIONS 88
by M. Selvakumar
The technique for automated web-user-interface testing presented here is based on HTML, JavaScript, and CGI, and implemented for Netscape Communicator 4.04 and Apache 1.2.

DR. DOBB'S JOURNAL (ISSN 1044-789X) is published monthly by Miller Freeman, Inc., 600 Harrison Street, San Francisco, CA 94017; 415-905-2200. Periodicals Postage Paid at San Francisco and at additional mailing offices. SUBSCRIPTION: $34.95 for 1 year; $69.90 for 2 years. International orders must be prepaid. Payment may be made via Mastercard, Visa, or American Express; or via U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year. POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80328-6188. GST (Canada) #R124771239. Canada Post International Publications Mail Product (Canadian Distribution) Sales Agreement No. 0548677. FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 30 Montgomery St., Jersey City, NJ 07302; 212-332-7100.

E N T S

MAY 1999
VOLUME 24, ISSUE 5

PROGRAMMER'S TOOLCHEST

THE VERSION CONTROL PROCESS 100
by Aspi Havewala
Source-code version control is a set of working rules for code sharing that lets developers modify files in an exclusive way. As such, it is one of the most important, yet least understood, areas of software development.

COLUMNS

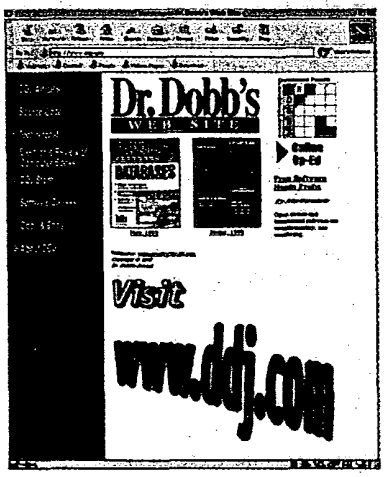
C PROGRAMMING 115
by Al Stevens
Al ponders the question, "What's in an *argv*?" and speculates on why the answer is different for DOS and UNIX developers.

JAVA Q&A 121
by Lou Grinzo
How do you run untrusted classes? Lou takes a look at a couple of different answers to this question.

ALGORITHM ALLEY 125
by Jon Bentley
Last month, Jon presented techniques for analyzing the performance of algorithms. This month, he examines how code-tuning techniques speed up the various algorithms.

DR. ECCO'S OMNIHEURIST CORNER 130
by Dennis E. Shasha
Dr. Ecco joins forces with the NSA, FBI, and other crime-stoppers to help fight web terrorism.

PROGRAMMER'S BOOKSHELF 133
by Gregory V. Wilson and William Stallings
Greg examines *Component Software: Beyond Object-Oriented Programming*, by Clemens Szyperski, while William takes a look at Neil J. Gunther's *The Practical Performance Analyst: Performance-By-Design Techniques for Distributed Systems*.



FORUM

EDITORIAL 8
by Jonathan Erickson

LETTERS 10
by you

NEWS & VIEWS 16
by the DDJ staff

OF INTEREST 142
by Eugene Eric Kim

SWAINE'S FLAMES 144
by Michael Swaine

RESOURCE CENTER

As a service to our readers, source code and related files, and author guidelines are available at http://www.ddj.com/. Source code is also available via anonymous FTP from ftp.ddj.com/ (199.125.85.76). Letters to the editor, article proposals/submissions, and inquiries can be sent to editors@ddj.com/, faxed to 650-358-9749, or mailed to *Dr. Dobb's Journal*, 411 Borel Ave., Suite 100, San Mateo, CA 94402-3522.

For subscription questions, change of address, and orders, call 800-456-1215 (U.S. or Canada). For all other countries, call 303-678-8475 or fax 303-661-1181. E-mail subscription questions to ddj@neodata.com/ or write to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188.

Back issues may be purchased for $9.00 per copy (includes shipping and handling). For issue availability, send e-mail to orders@mfi.com/, fax to 785-841-2624, or call 800-444-4881 (U.S. and Canada) or 785-838-7500 (all other countries). Back issue orders must be prepaid. Please send payment to *Dr. Dobb's Journal*, 1601 W. 23rd Street, Suite 200, Lawrence, KS 66046-2700.

Individual back articles may be purchased electronically at http://www.ddj.com/ as ZIP archives.

NEXT MONTH

In June, we examine the topic of object-oriented design, and announce the recipients of the *1999 Dr. Dobb's Excellence in Programming Awards*.

Object Persistence
Beyond Serialization

Increasing productivity and reducing maintenance

Timo Salo, Justin Hill, Scott Rich, Chuck Bridgham, and Daniel Berg

Most commercial high-volume databases are based on either the relational or service paradigm (that is, databases encapsulated within transaction processing monitors). Persisting objects in these nonobject-oriented databases is a major challenge when building large-scale applications.

On a small scale, object persistence is easy to solve. Serialization, for example, has been presented as a method for providing simple object persistence. However, scaling up introduces a new set of requirements. Many enterprise object systems involve object models with complex inheritance hierarchies and large numbers of object relationships. The runtime configuration often includes multiuser databases that can be both relational and nonrelational. The object model and database model are often designed by different groups of people, therefore requiring a loose coupling between the models. The design of a scalable object persistence framework must adequately address issues related to performance with complex object models, support for complex object transactions, transformations from object inheritance structures and associations to native database structures, translating object queries to native database queries, and accessing objects across multiple database paradigms.

There are several standards and specifications related to object databases and object persistence, including the Object Management Group (OMG) Standard, Object Database Management Group (ODMG) Standard, and Enterprise JavaBeans (EJB) Specification. However, none of these specifications address the actual implementation of a persistence engine. At best they describe interfaces and high-level components that form the API of the system.

In this article, we'll describe techniques and frameworks required to successfully implement scalable object persistence for complex systems. We'll address topics such as required

(continued on page 22)

The authors are software engineers working in IBM's VisualAge Features Development group. They can be contacted at tjsalo@us.ibm.com.

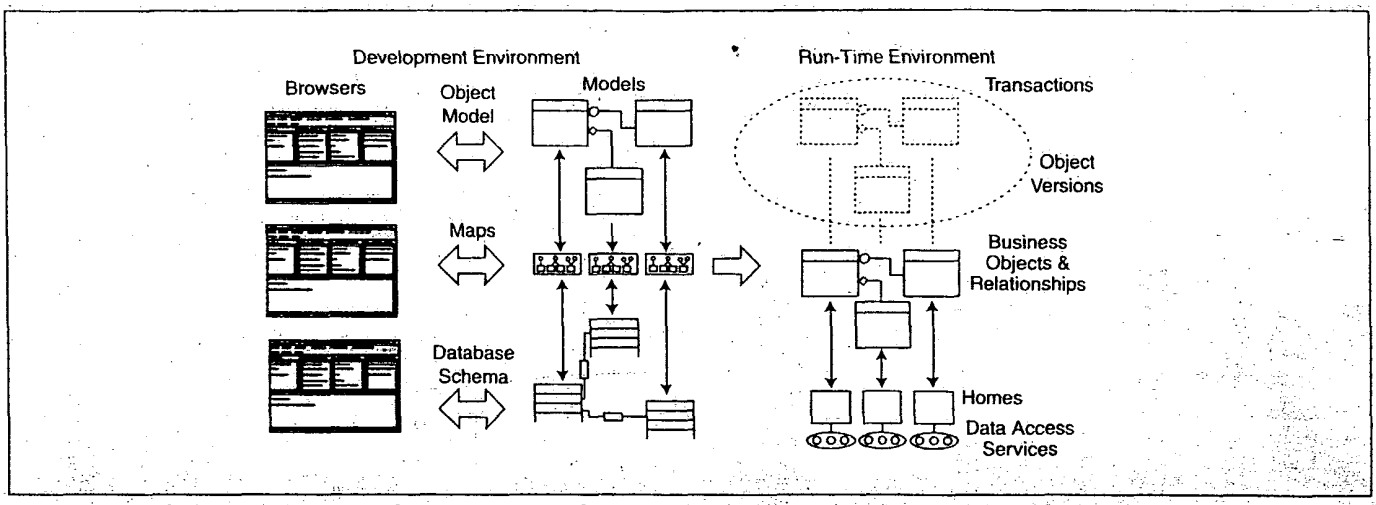


Figure 1: *High-level architecture for a persistence framework.*

(continued from page 19)
metainformation, read-ahead and caching, queries, object associations, and concurrent and nested transactions. We have pioneered these techniques for almost 10 years in many large-scale projects. Various aspects of the technology we describe have been incorporated in IBM development tools, including VisualAge for Java (Persistence Builder), VisualAge for Smalltalk (ObjectExtender), and EJB development tools for WebSphere.

General Architectures

Persistence frameworks typically consist of two high-level components: the development-time toolkit and the run-time persistence engine. Figure 1 is an example of high-level architecture for a persistence framework.

The development toolkit usually includes tools for collecting metainformation about the object model and database, and tools for generating business object classes and database queries.

There are two approaches for implementing the run-time engine. One approach is to have the metainformation available at run time, and generate the queries for retrieving objects on-the-fly as the application traverses various object relationships. This approach makes it possible to build dynamic, flexible applications that have no navigation restrictions within the object model. However, the amount of memory used by the metainformation and run-time query generation usually results in poorer performance. Another approach is to generate the queries at development time. Little explicit metainformation is needed at run time with this approach. Execution of the generated queries is faster, because run-time inferencing is not needed and the queries can often be optimized for the database. The drawback is that the object model traversal paths are fixed. If more paths are needed, more queries need to be generated and compiled.

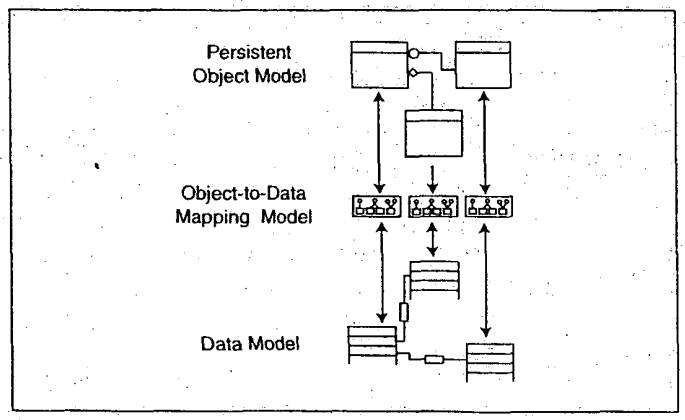


Figure 2: *Relationships between various metamodels.*

Metainformation

Metainformation for an object persistence framework includes information about the application's object model, the target database's data model, and the queries needed to service the application. As Figure 2 shows, the information is often grouped into the following models:

- The data model for describing the relevant subset of the database schema.
- The persistent object model for describing the persistent components of the business domain model.
- The mapping model for describing the mapping between the object model and the data model.

How much detail is captured and whether the metainformation is partitioned in one large model or various separate submodels depends on issues of flexibility, efficiency, and expressiveness. Therefore, there is no single correct way to package the information, but all the following must be captured in some form somewhere in the framework.

The data model represents the logical view of the database. It is a subset of the tables, views, and columns in the database schema that are relevant to object systems. This includes information on entity qualifier names, logical and physical names of entities, column datatypes, and conversions from database types to object language types. Further refinements could include information on database column functions such as sums and averages.

The data model can be augmented with information that is not explicitly kept in the database schema. For instance, the relationships implicitly defined by the foreign-key references in the schema can be modeled as first-class connection objects in the data model. Enhancing the data model with connections makes the mapping of object associations to database

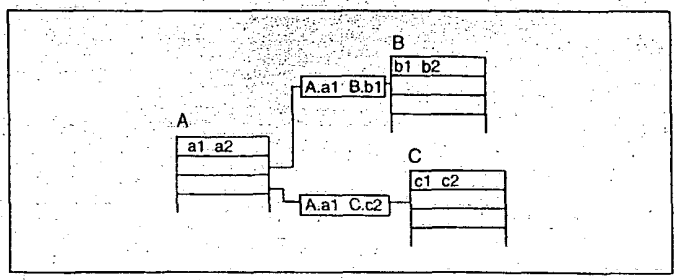


Figure 3: *A structural data model.*

(continued from page 22)

relationships a significantly simpler task. Figure 3 presents an enhanced data model (the structural data model).

It is not absolutely necessary to have a separate data model. However, without such a model, much of this information must be captured in the mapping model, thus overloading its behavior and state.

The persistent object model is a subset of the application's object model. It represents only the portion of the business object model that requires persistence behavior. It can be a subset of classes within the complete business object model and a subset of the instance variables within a single class. The allowed types for the attributes can also be captured for validation purposes.

Besides modeling the simple attributes, the associations between the persistent objects can also be modeled. This makes the object model independent from the mapping model, allowing a clear mapping between the foreign-key relationships in the data model and the object associations in the persistent object model.

The definition for the object identifiers can be captured in the object model rather than in the mapping model, again allowing simple mapping between the primary key column(s) in the data model and object identifier in the object model.

The persistent object model is optional, and much of the information that it provides can be held in the mapping model. However, without the object model (as well as without the data model) there is a risk of overloading the behavior and state of the mapping model.

The most minimal system that would be of any interest requires at least a model of mapping between the object structure on one side and the target database structure on the other. The mapping model contains the essential instructions to the system of where the data retrieved from the database is to be placed in the objects. The mapping model must define which object class corresponds to which table and which object attributes correspond to which columns. Refinements could include mapping one object to multiple tables and one instance variable to multiple columns, conversions of column data from primitive types to higher-level object types, and defining which columns act as database-conflict-detection predicates. Figure 4 shows examples of class-to-table mapping schemes.

If associations are to be supported transparently, then the mapping must also define which foreign-key relationship corresponds to which object association in the object model. Figures 5 and 6 illustrate various relationship-mapping schemes.

Finally, if inheritance is supported then the mapping model should capture all such information. This would include the type of inheritance employed in the database, type discriminator values for choosing the appropriate class, and/or foreign-key relationships between tables. Figure 7 shows examples of inheritance mapping schemes.

Cache

Various read-ahead and caching strategies can improve a persistence framework's efficiency and flexibility. Without read-ahead and caching capabilities, the application is always starved for data, parsimoniously reading from the database as associations in the persistent object model are traversed and bringing back data only one level at a time. With an object model that has many relationships, this can cause a large number of expensive database roundtrips.

A read-ahead scheme lets the application minimize the number of database roundtrips by retrieving large object composition

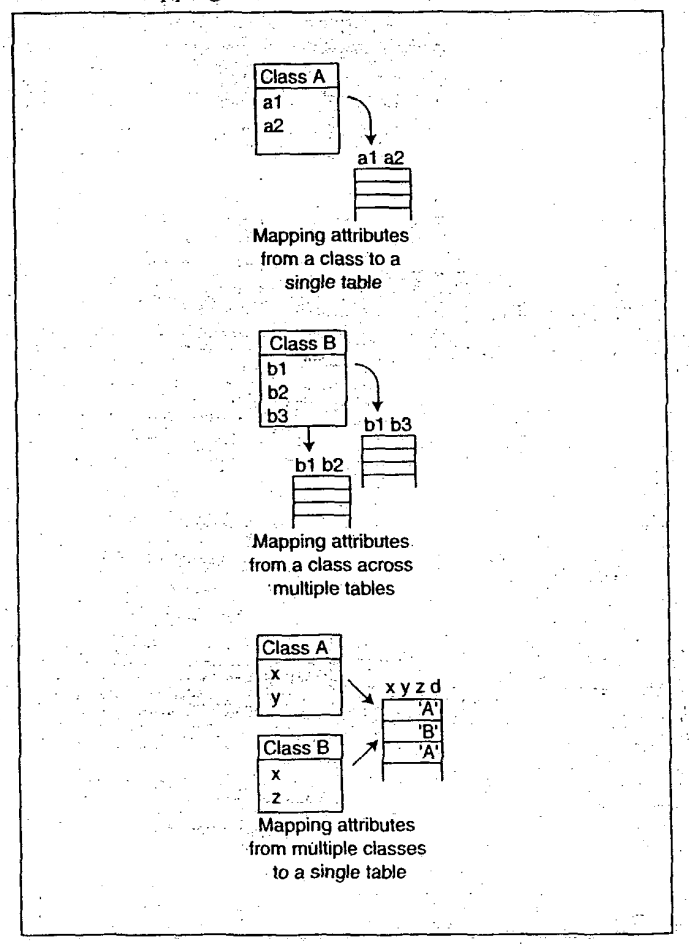


Figure 4: Various class-to-table mapping schemes.

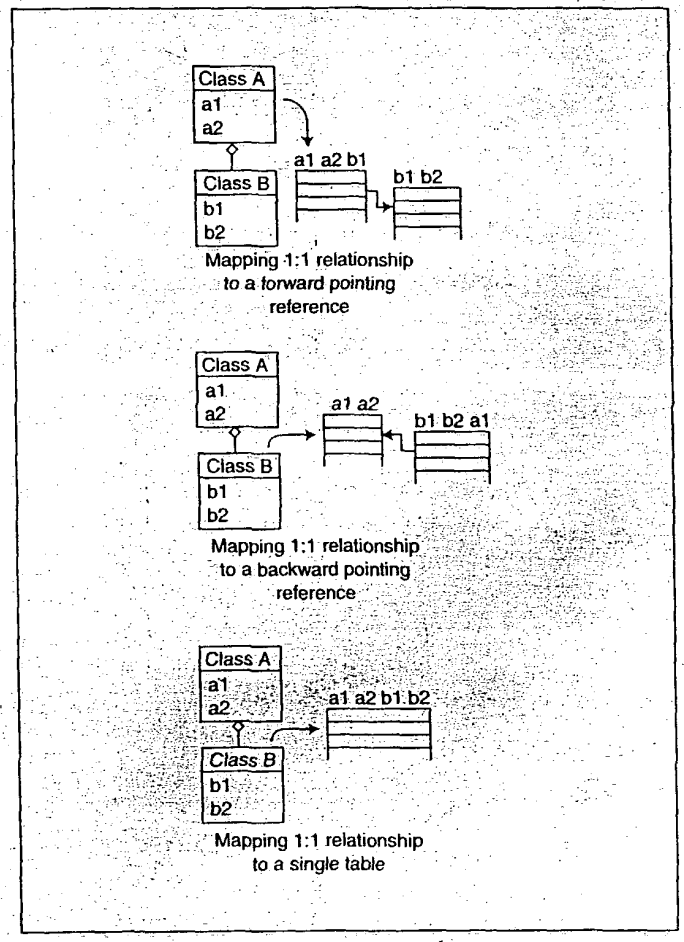


Figure 5: 1:1 association mapping schemes.

trees within one query. Read-ahead involves instantiating the requested objects and caching the data for their related objects, thereby making sure that the data is present for the objects that are most likely needed next by the application. How far ahead objects are read is determined by application requirements. Flexibility is gained as the queries can be tuned without affecting the structure or workflow of the application.

Reading objects ahead often results in too much data. Therefore, it is desirable to keep the data in binary format to delay or avoid the performance cost of instantiating unused objects. Instantiation of persistent objects is then performed in two stages: First, the data is brought into the cache, then the objects are instantiated from the cache upon demand. Leaving the data in a form that is smaller than a fully instantiated object saves space as well.

The key to implementing the read-ahead feature is to extend the caching scheme to include the relationship semantics of the underlying database. Database queries have fixed access paths that may differ from the object model navigation order. Therefore, the data in the cache must be organized in a fashion that allows dynamically composing any access paths defined in the database. In the case of relational databases, this means that the foreign-key references are extracted from the result set and maintained in a structured data cache. Figure 8 shows a structured data cache.

Registry

To guarantee the uniqueness of the objects within the application's memory, each instantiated persistent object must be registered into a centralized registry. The objects are usually identified in the registry using their persistent object identifiers; see Figure 9.

As Figure 10 illustrates, when an object is retrieved using its object identifier the registry is searched first, then the data cache, and finally the database. The registry can be global if it is implemented using weak pointers, because objects are automatically removed from the registry when other objects no longer reference them. However, if weak pointers are not available, the registry must be localized. For example, transactions provide a good scope for local registries.

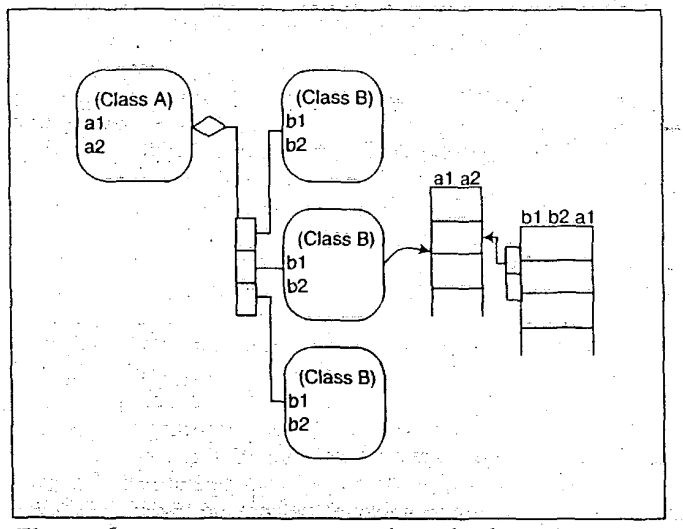


Figure 6: *1:m association mapped to a backward pointing foreign-key reference.*

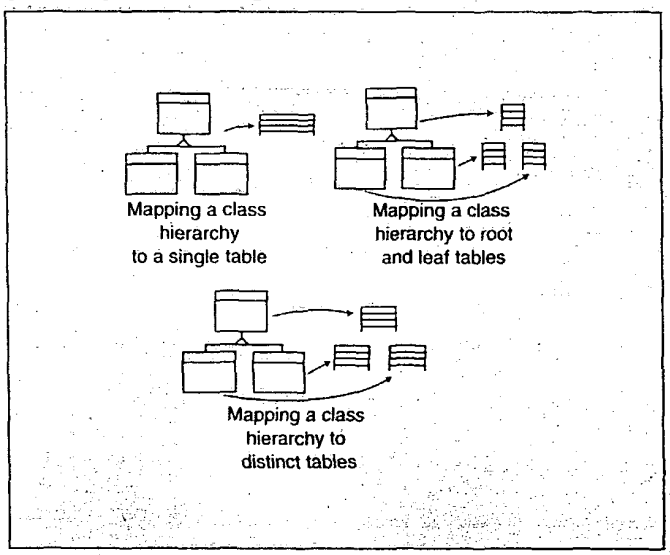


Figure 7: *Inheritance mapping schemes.*

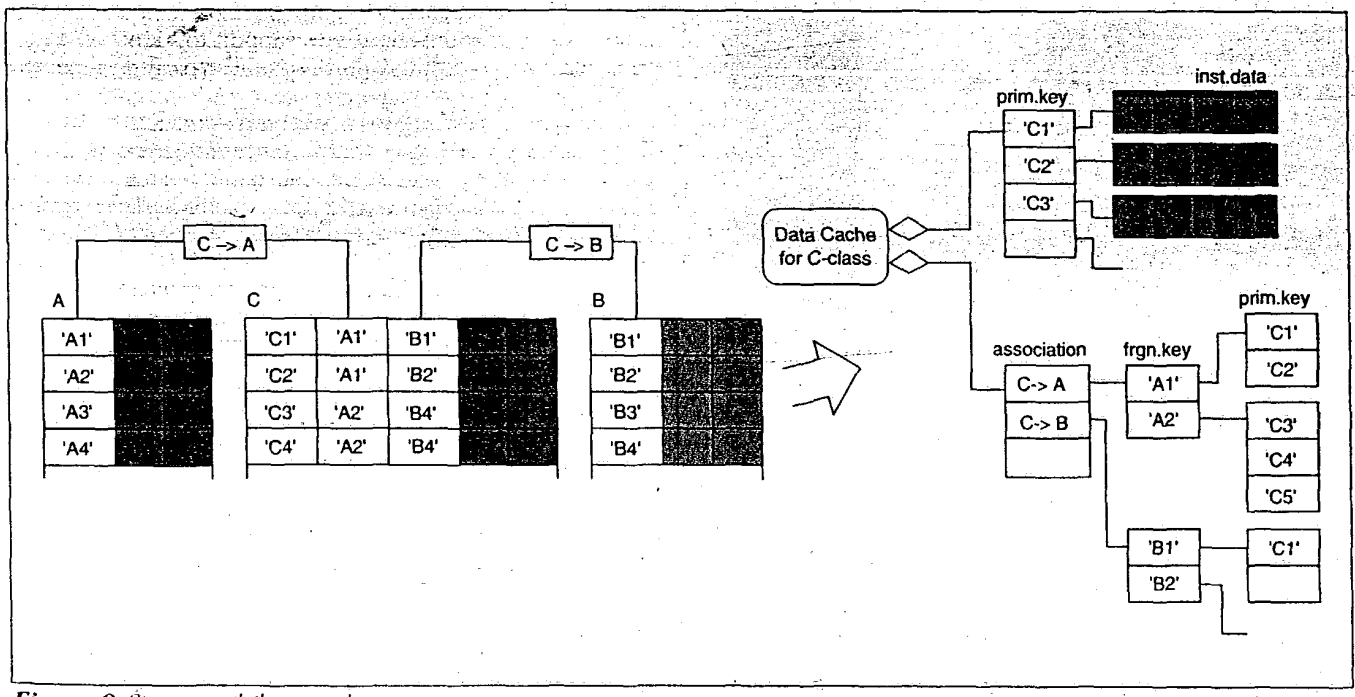


Figure 8: *Structured data cache.*

Qu ries

From the persistence framework's point of view, queries are the behavior of persistent objects on their target database. Query in this context means any operation supported by the target database and executed by the persistence framework. This includes basic create, read, update, and delete operations; inquiries (does an entity exist in the database, the sum of a set of columns); and specific operations defined by a particular database server such as "balance the account."

Invocation differences between different target datastores include details such as native query representation, error handling, and result data interpretation and processing. The native query representation typically can be strings (as with dynamic SQL), host variables (static SQL, stored procedures), or records (mainframe messaging).

Encapsulating the native query details within query objects can standardize target database invocation. For instance, an object application would never know whether the query object contains a SQL string, or invokes a stored procedure or a message to a mainframe transaction-processing monitor. Figure 11 presents two sets of encapsulated queries targeting two different types of datastores.

Queries can be grouped into two broad categories—write queries (SQL insert, update, and delete, for example) and read queries (SQL select).

Input for write queries can be either keys (for instance, delete an object based on its key) or full objects (insert an object); either of which can be collections. Queries targeting relational databases operate on a single object. Queries targeting stored procedures or mainframe transaction-processing monitors usually take multiple objects as input parameters.

Write queries extract the data from persistent objects and convert it to the target database form. Depending on the datastore, the data is placed into a query string, a query's host variables, or a record structure. In the case of nested records (mainframe messaging), the data may also need to be recomposed according to the nesting structure; see Figure 12.

Because relational write queries can operate only on one object at a time, the number of database roundtrips within a complex transaction often becomes high. A useful performance optimization is to group the native queries together, then send them to the database as one package at the end of the transaction. Many relational databases support this kind of "batch" behavior. For procedure calls this is the typical mode of operation.

Read queries fall into two categories—those that have no scope limiting conditions ("all instances" queries, for example) and those that require parameters for search conditions ("finder" queries). Read queries that require parameters must address the same data conversion and recomposition issues as the write queries.

Restructuring the resulting data is necessary when the data is not shaped along object lines and/or the result contains data for

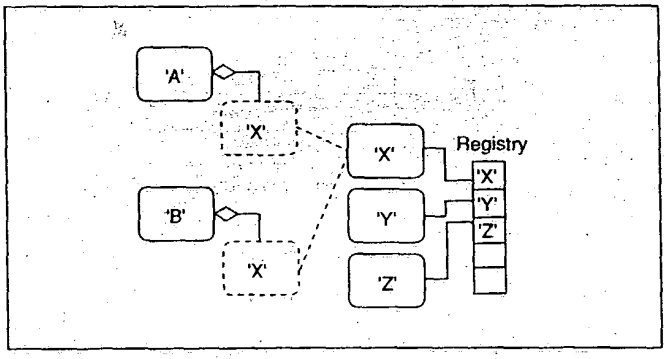


Figure 9: *Making objects unique using a registry.*

more than one kind of object. For example, queries involving certain inheritance strategies or reading ahead trees of objects require joins and unions that result in tuples containing data for multiple objects. A useful abstraction for the result processing is a data extractor. The data extractor contains all the necessary logic to extract, convert, validate, and compose the data into a form suitable for the target persistent object. In case of relational joins, the extraction logic must also eliminate redundant entries in the result set; see Figure 13.

To optimize the number of database roundtrips, the read queries need to be capable of loading trees of objects rather than reading one object at a time. The required native operations for relational queries are equijoin for loading chains of objects, unions and set differences for loading trees, and left-outer-joins for loading trees that allow missing leaves.

Various read-ahead and caching strategies can improve persistence framework's efficiency and flexibility

Associations

Describing the associations between object classes is an essential element of object modeling and design. UML and other object modeling methodologies provide ways of defining the semantics of associations in terms of their cardinality and navigability.

The behavior of associations can be fairly complex. The implementation details can be hidden behind accessor methods (*get* methods). Accessors for one-to-one associations return the member object of the association. An accessor for a one-to-many association returns a collection of member objects. Another approach (see Figure 14) is to implement associations as first-class objects (in-place association instances, proxies).

At run time, the object referential integrity should be maintained according to the semantics specified in the objects model, while allowing the application programmer the easiest and most flexible interface to the relationships. Mutators (*set* methods, for example) and collection add/remove methods should automatically invoke the appropriate referential integrity maintenance behavior, such as updating the inverse association.

Associations are especially important for persistent objects mapped to relational databases because associations can also provide automatic means for maintaining the database key referential integrity. When connecting persistent objects, the association will determine which persistent object holds the foreign key and update it appropriately with the primary key of the other object. Manually coding the database key maintenance is error prone and can easily lead to unmaintainable code. Figure 15 illustrates automatic maintenance of object and database key referential integrity. In this example, an employee object is automatically removed from its old department when the object is added to a new department. Also, the inverse relationship from the employee to the department is updated automatically.

Associations provide a semantically meaningful way for controlling the retrieval of objects from the database. As the application traverses associations, the related objects can be retrieved accordingly. Depending on the association, it is sometimes also desirable that traversal of one association triggers the retrieval of an entire graph of related objects. However, this kind of object graph read-ahead behavior requires advanced querying and caching techniques as described in the previous sections.

Translation from the object associations to the native database relationships may be very complex (see Figure 16). Simple relationship between two classes often translates to multiple relationships between multiple tables when inheritance is involved.

Transactions

In enterprise environments a single server application may serve multiple concurrent client transactions, each accessing an overlapping set of objects.

Many enterprise applications that reflect complex business processes (see Figure 17) also require that users can navigate freely between different views of the user interface, work with the result of uncommitted changes across views, and commit or cancel work that has been done on a view and on all subviews opened in a nested fashion. In short, the nature of complex multiuser enterprise applications requires that objects can be accessed from multiple concurrent and nested transactions.

To ensure the consistency of concurrently running transactions they need to be isolated from each other. The two methods for isolating the transactions are the conflict avoidance scheme ("pessimistic" scheme) and the conflict detection scheme ("optimistic" scheme). Which one to use depends on the type of transaction. Transactions that have a high penalty for failure should do whatever possible to prevent the failure

(continued on page 30)

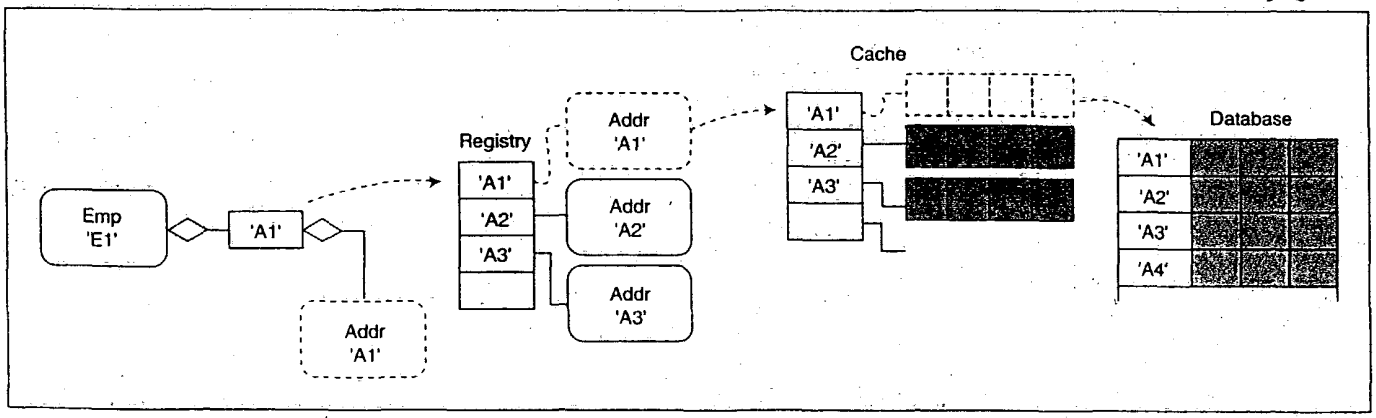


Figure 10: *Search sequence when retrieving objects.*

(continued from page 27)
(by explicitly locking the resources as early as possible). With low penalty transactions it is often worth trading the risk of failure to gain efficiency by using a conflict detection scheme.

The objects are copied from the database into the application's memory, where they may be held for extended periods of time. Therefore, the transaction isolation actually consists of two components: the object level isolation within one application, and the database level isolation across multiple applications. Both isolation components address multiuser issues, because one server application may also serve multiple clients, as in Figure 17.

The conflict avoidance scheme for GUI-driven, long-running transactions is usually unacceptable from a performance perspective. A conflict detection scheme where each transaction has a version of the concurrently accessed objects provides significantly better performance. However, managing multiple versions of the same object can be fairly complex.

One approach for implementing an object versioning mechanism is to divide business objects into two parts: a wrapper and a version (for example, an *EJBObject* and an *EntityBean*). When any object refers to a business object, it actually refers to its wrapper. The wrapper delegates the method invocations to the appropriate version, which contains the object's business behavior and instance data. When a business object is first accessed (*get/set* a property) within a transaction, a new version of the object is added to the current transaction's local registry. The new version is based on the version in the parent transaction's registry. Figure 18 shows multiple object versions within a tree of nested transactions.

Upon commit, the versions in a child transaction's registry are merged with its parent transaction's corresponding versions. If the transaction is a top-level transaction, the versions are also written into the database. The logic for detecting and resolving conflicts on merge is highly application dependent. The test may be as simple as comparing parent and child version numbers in

```
update address set streetno=34, ... where custno=456 and streetno=56 ...
```

Example 1: *Update statement with conflict detection predicates.*

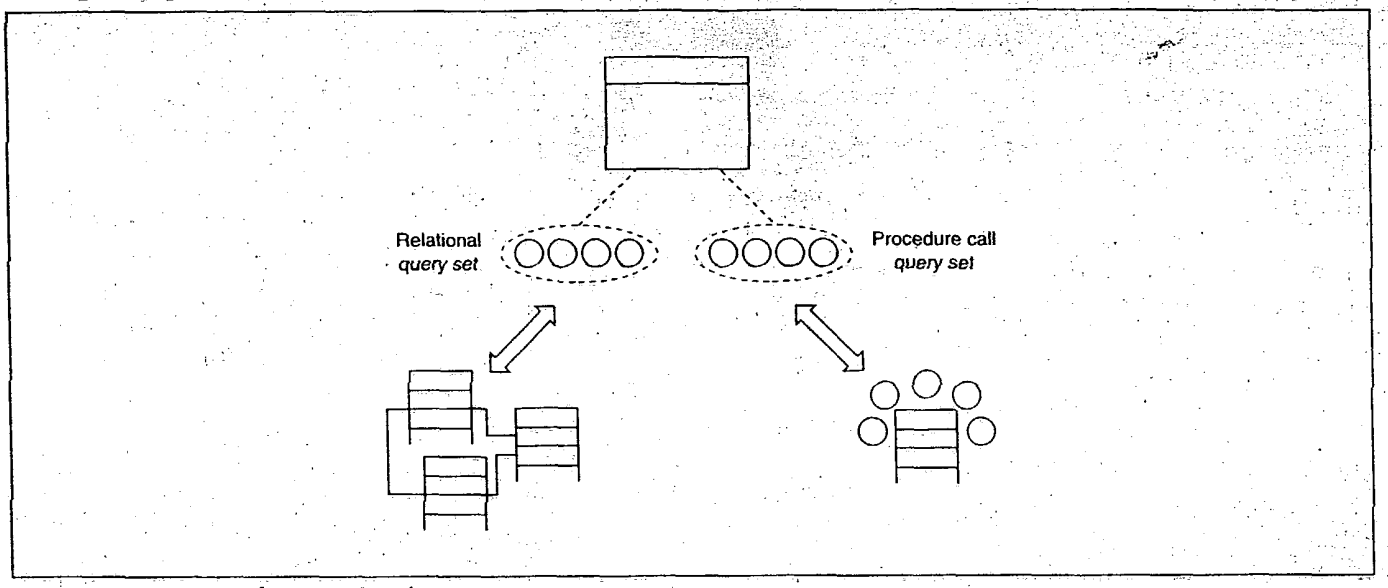


Figure 11: *Two sets of encapsulated queries.*

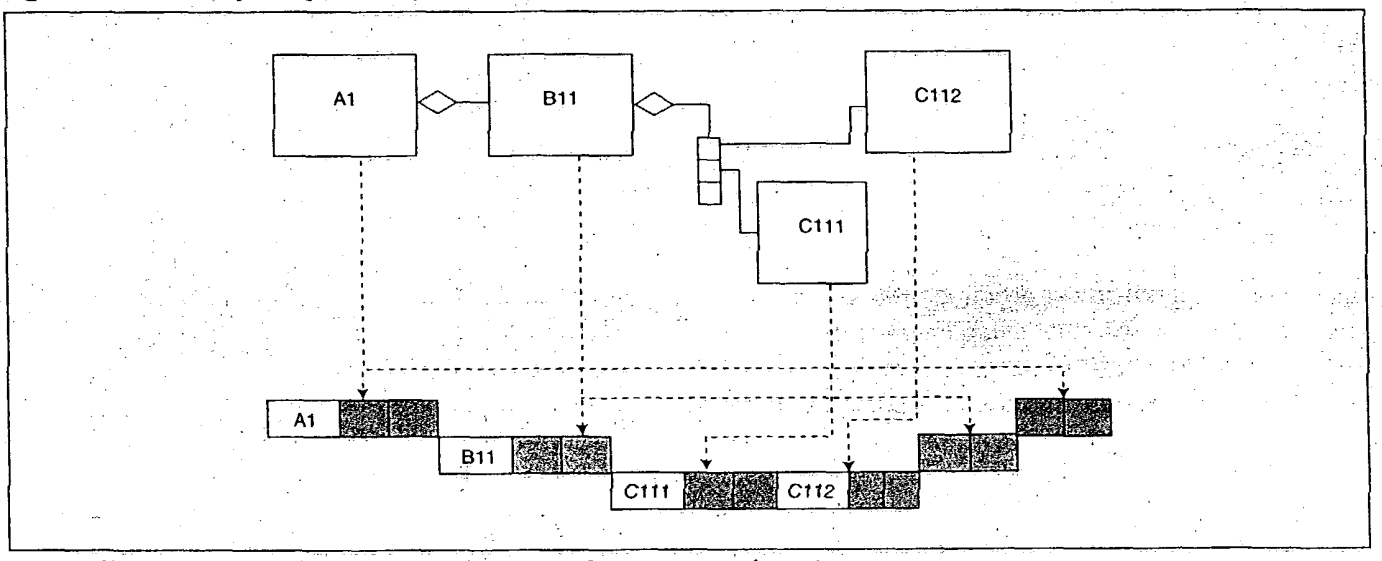


Figure 12: *Recomposing object instance data according to a record-nesting structure.*

order to determine if the parent version has been changed after the child version was created. For more advanced application-dependent testing the wrapper could have a conflict resolution call back method.

On rollback the child versions are simply dropped instead of having to restore object states in the parent transaction. After rollback there is no trace that either the child transaction or the child versions ever existed.

Many relational databases provide little support for row-level conflict avoidance. With most databases the row-level locking is available only in conjunction with cursors. However, cursors may be of little use for an object application that is accessing and holding onto large numbers of different types of objects in a random fashion. One trick for acquiring a row-level lock without a cursor is to touch a corresponding row (update a column without changing its value, for instance) when an object is first accessed within a transaction. If the row is already locked, the desirable action is often to raise an exception instead of waiting for the lock to be released.

Serialization has been presented as a method for providing simple object persistence

As with object level isolation, the logic for detecting and resolving database conflicts is application dependent. The two common conflict detection methods are either to reread and compare the database row to the modified object, or to add collision detection predicates (a set of attributes that constitute a conflict) to the *where* clause of the database *update* statement. Example 1 demonstrates conflict detection predicates. The *update* statement will fail if another user has changed the street number from its old value.

Rereading and comparing rows is expensive and should be used sparingly, because it requires multiple database roundtrips—locking, reading, and updating the row. On the other hand, the use of conflict detection predicates is lightweight and works fine in most situations. More sophisticated detection schemes can be composed of combinations of the aforementioned commands.

Most commercial databases have referential integrity (RI) constraints for maintaining the consistency of the database. These constraints require the database's store and delete operations to be executed in a specific order. This order does not necessarily match the order in which the objects are created or deleted within an object application. Furthermore, the database RI constraints do not map to the logical object associations in a consistent way. RI rules are enforced based on the foreign-key references, which may have more than one possible transformation when mapped to object associations. Manually coding the operation ordering is time consuming and error prone, easily leading to unmaintainable code. It is preferable to defer execution of the operations and let the transaction automatically decide the ordering upon its commit.

The ordering algorithm utilizes the information of how the object associations are mapped to the primary-key/foreign-key column pairs in the database, and the integrity rules defined for the key columns. For each object within the transaction, the algorithm iterates over the associations the object has with other objects. For each association, the algorithm tests if the object has either insert precedence (if the object is to be inserted) or delete precedence (if the object is to be deleted) over the association. If the object has a higher precedence, it will be moved accordingly in the transaction's participant list. Due to the nature of relational RI constraints, the algorithm remains fairly simple, because there cannot be circular constraints defined in the database (otherwise it would be impossible to insert a row that has a prerequisite to its own prerequisite).

API

From the programming and maintenance point of view, the number of persistent constructs that appear in the application code should be kept as low as possible. Having a low number of persistence constructs introduces minimal intrusion upon

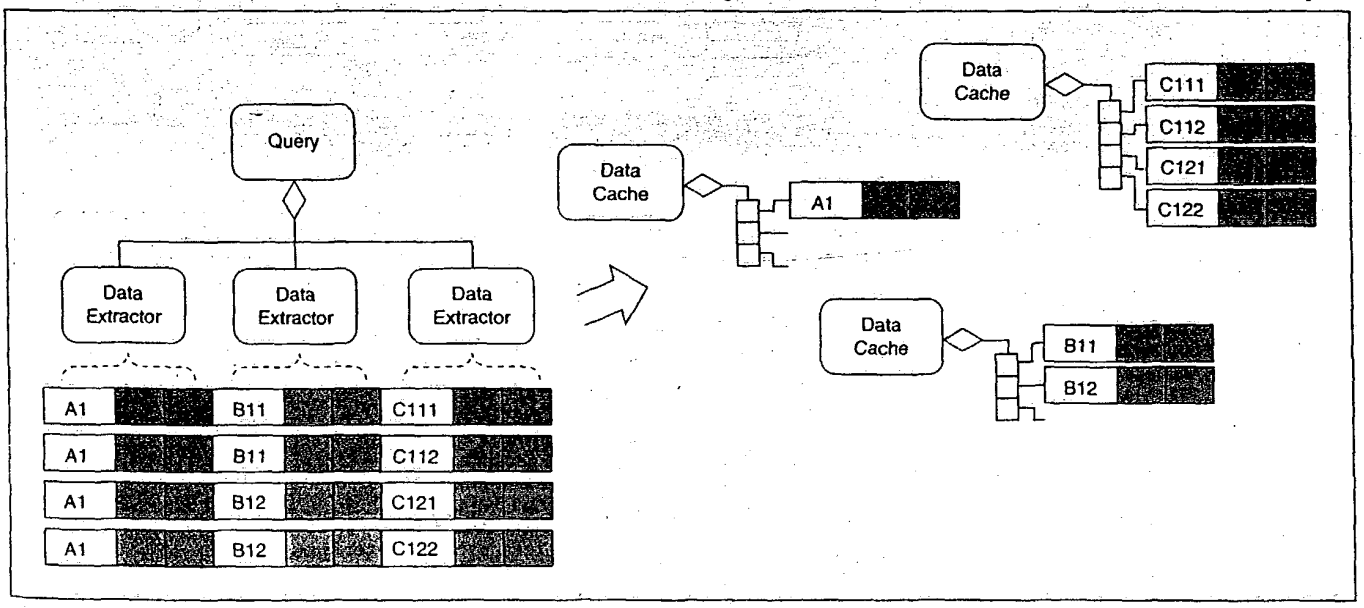


Figure 13: *Restructuring a relational result set and eliminating redundant entries.*

the application, thus allowing the database and application to remain loosely coupled. This loose coupling between the database and application lets you design an object model that models the application domain as opposed to modeling the database design and vice versa. The persistent framework must be intelligent enough to perform many of the necessary persisting processes automatically, without instruction from the application. Implementing persistent constructs as first-class objects and providing some of the persistence metainformation at run time are two of the keys that make a successful persistence framework. The interfaces provided by the persistence API can be grouped into the following categories:

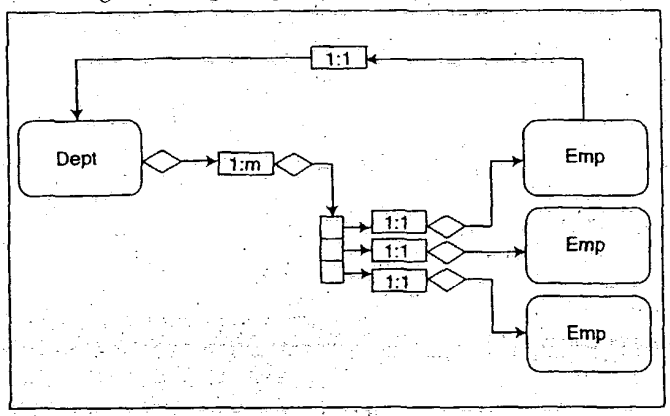


Figure 14: Associations implemented as first-class objects.

- Business object interface. Protocol for accessing attributes from the business object.
- Life cycle interface. Protocol for creating and destroying business object instances.
- Finder interface. Protocol for finding business object instances.
- Transaction interface. Protocol for creating, committing, and rolling back transactions.

For example, the Enterprise JavaBeans (EJB) Specification defines interfaces that correspond to these categories. The remote interface for entity Beans corresponds to the business object interface. The EJB home interface has the same responsibilities as the life cycle and finder interfaces. The transaction interface is provided by the *UserTransaction* in the Java transaction package, which is one of the prerequisites for the EJB.

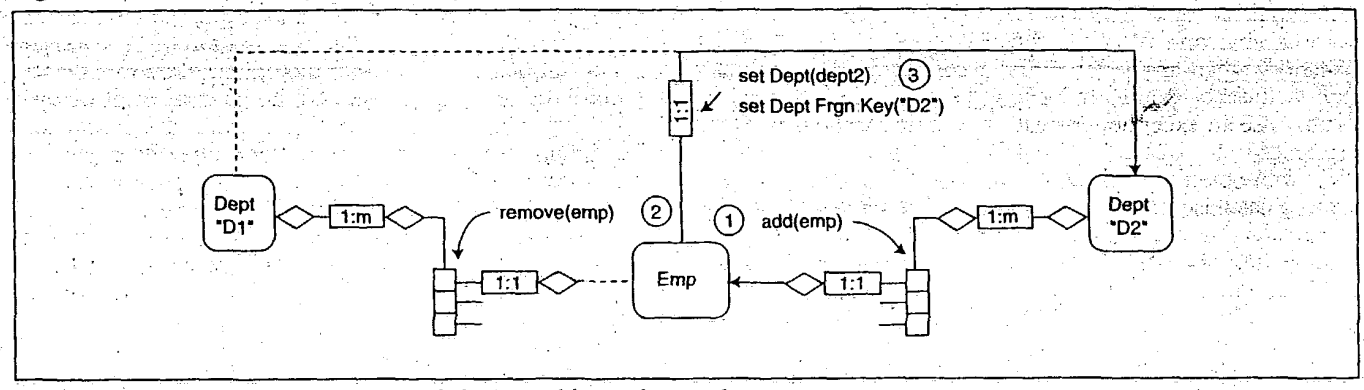


Figure 15: Automatic maintenance of object and key referential integrity.

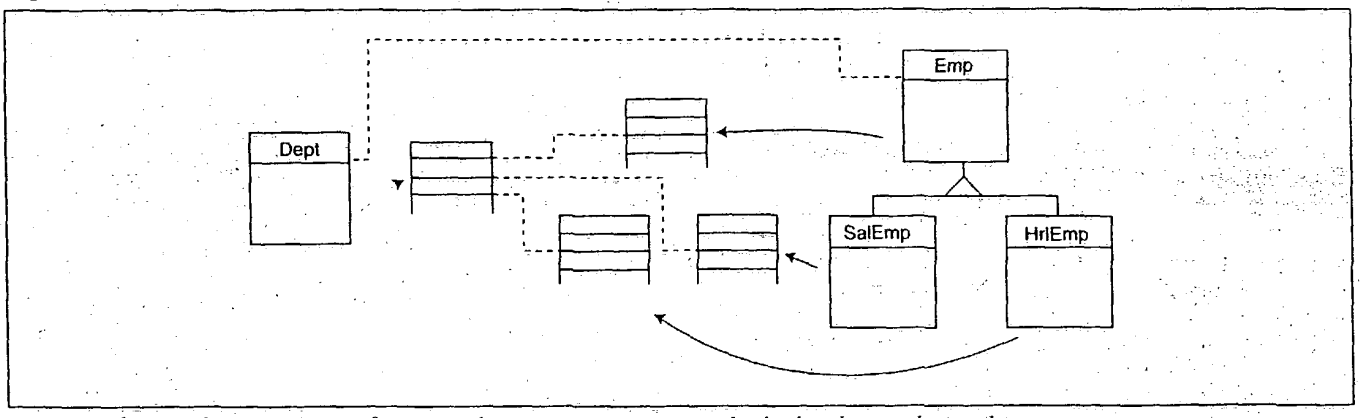


Figure 16: Complex translation from an object association to multiple database relationships.

```
Transaction tx = Transaction new();
EmployeeHomeImpl empHome = EmployeeHomeImpl.singleton();
Employee emp;
AddressHomeImpl addrHome = AddressHomeImpl.singleton();
Address addr;
tx.begin();                              //begin a new transaction (transaction interface)
emp = empHome.findByKey("1234");         //find an employee instance (finder interface)
addr = addrHome.create();                //create an address instance (factory interface)
addr.setStreet("123 Somewhere Dr.");     //set attributes of the address (bus.object interface)
 ...
emp.setAddress(addr);                    //set employee's address (bus.object interface)
tx.commit();                             //commit the changes (transaction interface)
```

***Example 2:** Sample persistence API code.*

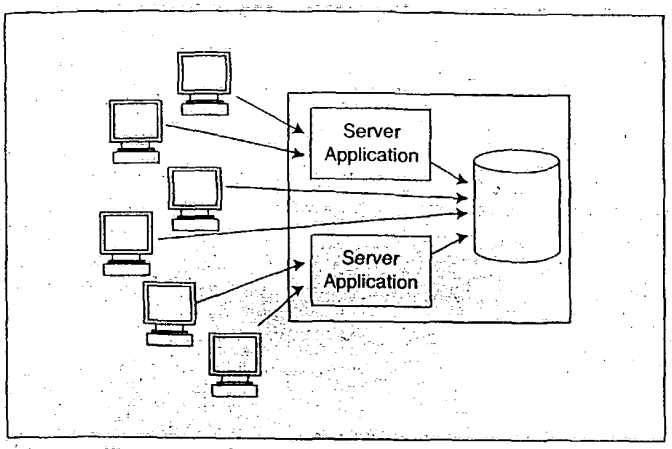


Figure 17: A typical system configuration in an enterprise environment.

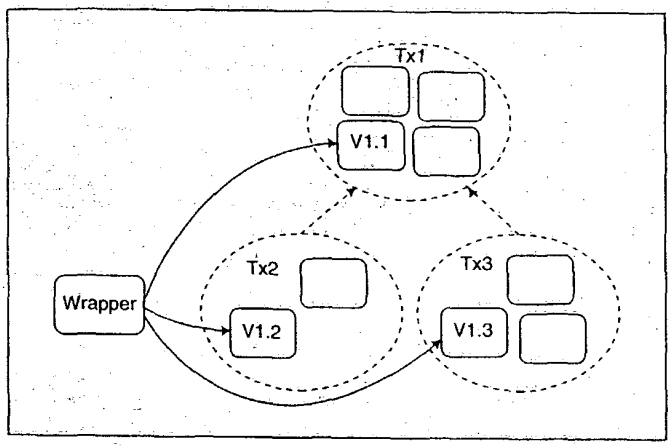


Figure 18. *Multiple object versions within a tree of nested transactions.*

Example 2 demonstrates the use of the persistence API by retrieving an employee object, creating an address object, associating these two objects together, and committing the changes to the database.

Conclusion

The rationale for building an object persistence framework are, of course, increased productivity and reduced maintenance costs. Independence between object applications and databases allows enterprises to develop and maintain more complex applications and still leverage existing data management infrastructures.

Implementing a full-blown object persistence framework easily represents several years worth of work. The more flexibility and performance that is required from the framework, the more complex the framework becomes. Yet almost any framework is better than no framework. Even a simple framework can help in structuring the code in a clean and logical way. For example, the mapping metainformation can implicitly be represented as inlined code and the query objects can encapsulate handcrafted SQL strings. The areas worth spending more time in creating generic components, however, are the associations and the transactions because they have a direct impact on the application programming model. There are also several commercial object persistence frameworks available that are usually a viable alternative to in-house development, especially when the target application is complex and critical to the enterprise.

DDJ